

Symbol Proxy Client Sample

*This sample is compatible with the June 2015 Xbox One XDK or later.*

*Compatibility note: There’s a known issue in the June XDK where the xbSymbolProxy does not properly resolve source line information. This is fixed in the August XDK and will be included in the June XDK QFE 1*

# Description

This sample demonstrates how to use the SymbolResolve APIs to resolve symbol information for addresses on the console. There are currently two APIs in SymbolResolve.h, GetSymbolFromAddress() and GetSourceLineFromAddress(). These APIs are useful in console-side diagnostic code, for example to decode the return addresses in a back trace that you capture. See the XDK documentation for more details.

The sample also provides some useful utility code in CallStack.h/.cpp for capturing and decoding callstacks. Three helper functions for capturing back traces:

Capture the backtrace for the current thread

CaptureBackTraceFromCurrentThread()

Capture the backtrace starting with the provided context record.

CaptureBackTraceFromContext()

Count the number of stack frames in the callstack starting with the provided context record.

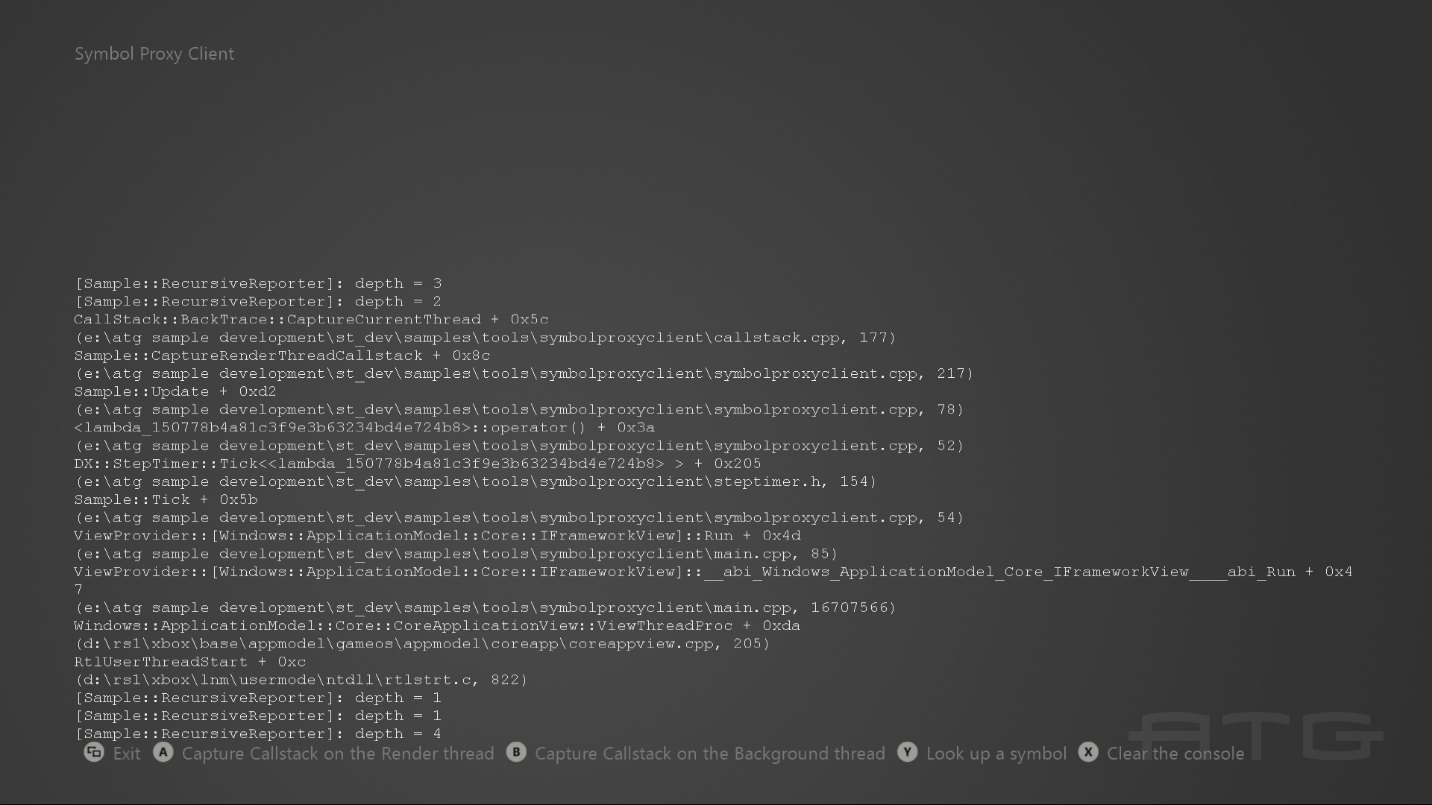
GetFrameCountFromContext()

Also in CallStack.h/.cpp is a BackTrace class that simplifies the process of capturing a back trace and resolving the symbols for the return addresses.

# Using the sample

|  |  |
| --- | --- |
| Action | Gamepad |
| Exit | View Button |
| Capture a call stack on the Render Thread | A Button |
| Capture a call stack on the background thread | B Button |
| Look up a symbol | Y Button |
| Clear the text console | X Button |

In order to resolve symbols, you must run xbSymbolProxy.exe on the development machine that has been configured to connect to the console using xbConnect.exe. If you do not, the sample will simply print an error when it cannot resolve any symbols.



# Implementation notes

**Resolving Symbols and Performance**

In order to resolve symbols, the API must communicate with xbSymbolProxy.exe running on the desktop machine. This is a slow operation and should be performed asynchronously on a background thread rather than on a performance-critical thread (such as you would do with any other slow operation.) To demonstrate this best practice, the sample performs all symbol resolution asynchronously.

The BackTrace class splits the capturing and resolution steps into two distinct operations. Use BackTrace::CaptureCrossThread and BackTrace::CaptureCurrentThread to capture the back trace – this is fairly performant and furthermore the performance cost cannot be avoided on the thread where you are capturing the back trace. Use BackTrace::Resolve() to communicate with the symbol proxy and resolve the symbol information for the addresses. This can be quite slow, but there is no requirement to resolve the symbols on the same thread were you captured the back trace – so you can safely call BackTrace::Resolve() asynchronously on a background thread.

**Suspending Threads**

You should not add or remove stack frames while you are trying to capture a back trace. This is easy to achieve if you are capturing the back trace on the current thread. However, if you are capturing a back trace on a different thread, you must use some mechanism to “freeze” that thread. The BackTrace class indirectly calls SuspendThread and ResumeThread before and after capturing the back trace. The calls to SuspendThread and ResumeThread are managed in a helper class called ThreadSuspender.

It is important to understand that when a thread is suspended, it is possible that it may own some kind of synchronization primitive(s), such as a critical section, mutex, or semaphore. This could result in a deadlock scenario in the case when the suspending thread needs to take ownership of the synchronization primitive before resuming the suspended thread. This is why it is generally not a good idea to use SuspendThread.

One case that is particular problematic occurs if the suspending thread needs to allocate memory while the other thread is suspended. If the suspended thread happens to be holding a lock on the heap, then this will cause a deadlock. In order to avoid such a deadlock scenario, the BackTrace class uses a global, static buffer in order to store the return addresses for the back trace and thus avoids the need to allocate any memory while capturing the back trace. The class sets aside enough space for 64 stack frames.